OF.CPP: Consistent Packet Processing for OpenFlow EPFL Technical Report (EPFL-REPORT-186714)

Peter Perešíni[†*], Maciej Kuźniar[†*], Nedeljko Vasić[†], Marco Canini[♯], and Dejan Kostić[‡]
[†]EPFL [♯]TU Berlin / T-Labs [‡]Institute IMDEA Networks
[†]<name.surname>@epfl.ch [♯]m.canini@tu-berlin.de [‡]dkostic@imdea.org
[*] These authors contributed equally to this work

ABSTRACT

This paper demonstrates a new class of bugs that is likely to occur in enterprise OpenFlow deployments. In particular, step-by-step, reactive establishment of paths can cause network-wide inconsistencies or performance- and space-related inefficiencies. The cause for this behavior is inconsistent packet processing: as the packets travel through the network they do not encounter consistent state at the OpenFlow controller. To mitigate this problem, we propose to use transactional semantics at the controller to achieve *consistent packet processing*. We detail the challenges in achieving this goal (including the inability to directly apply database techniques), as well as a potentially promising approach. In particular, we envision the use of *multi-commit transactions* that could provide the necessary serialization and isolation properties without excessively reducing network performance.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Network operating systems

Keywords

Software-Defined Networking, Consistency, Multi-commit transactions, ACID

1. INTRODUCTION

Software-defined networking (SDN), and OpenFlow in particular, is increasingly being deployed. Despite having access to a logically centralized view of the network, when using any of the popular controller frameworks (*e.g.*, NOX, POX, FloodLight) the network programmer uses the low-level OpenFlow interface. Thus, she still has to deal with the asynchronous, distributed nature of the underlying network.

In this paper, we demonstrate a new class of bugs that can cause network-wide inconsistencies or performance- and space-related inefficiencies. In the first example, the controller that installs rules on a switch-by-switch basis can create a forwarding loop after an event that alters the topology. In the second example, a controller running the learning switch application floods a packet toward the (unknown) destination host, and learns the destination's location when this responds to one of the flooded packets. The remains of the flood can then cause additional, unnecessary rules to be installed in the network, along which a packet storm might occur.[1]

These bugs can have serious consequences. Specifically, once created, the forwarding loop (first bug) stays until the rules expire. For the second example, when the rule amplification problem manifests itself, on average 30% more rules get installed in our experiments. The excess rules can decrease the network performance as a whole due to limited TCAM space in the switches, and sometimes limited intra-switch bandwidth for new rules [4].

The underlying cause for the problem is inconsistent packet processing; as the packet(s) traverse the network they do not encounter consistent state at the controller (after being forwarded by switches with no rules for them). In the learning switch for example, it is the response from the host that changes the state in the controller in a way that negatively affects the remaining flooded packets.

Although this problem may not manifest in environments that are running a proactive controller, it is nevertheless important. First, it is likely to occur in enterprise OpenFlow deployments where there is less of a chance of having a thoroughly maintained host map. Second, supporting zero-configuration requires a reactive controller. Third, administrators often insist on having control over the flows, to implement *e.g.*traffic redirection through middleboxes. The existence of hybrid networks and/or multiple controllers can also prevent installation of rules on entire paths. Finally, some controllers have to retain the ability to act in a reactive manner because there might not be enough rule space (either wildcarded or exact match).

Recent work in this space addresses some issues with the low-level OpenFlow API. For example, Monsanto *et al.* [12] propose programming language techniques that raise the abstractions for managing SDNs and Reitblatt *et al.* [13] propose general mechanisms for managing network updates. However, they do not solve the problem described here. Specifically, consistent updates [13] deal only with the state in the switches, whereas the inconsistent packet processing

[1]We refer to this problem as the "rule and packet amplification" problem (rule amplification for short).

Extended version of the the HotSDN'13 paper.

problem has to do with the state at the controller itself.

To mitigate this problem, we propose to use transactional semantics at the controller to achieve *consistent packet processing*. However, we do not advocate a straightforward application of the classic ACID (Atomicity, Consistency, Isolation and Durability) semantics. Numerous issues discourage its use, including: (*i*) lack of clear specification of the events pertaining to a single transaction, (*ii*) difficulty in knowing when to commit (*e.g.*, tracking when the last packet exits the network is difficult and costly), and (*iii*) difficulty of rolling back network state and packets already sent from the network's edge.

To achieve the desired isolation properties in the domain, we envision the use of *multi-commit transactions*. The main idea is to allow each event (*e.g.*, `PacketIn` processing at the controller) to run as a subtransaction and commit, while a (high-level) transaction continues to exist. We carefully examine the possible orderings of state accesses and updates caused by concurrent subtransactions, and specify when they are allowed to commit. The (high-level) transaction still consists of all events related to a particular packet. The rule amplification problem for example would be avoided, as all packets in a flood interact with the controller in isolation, as if there are no other flows in the network.

The isolation properties provided by multi-commit transactions are not complete. In particular, newer transactions see the effect of committed subtransactions of older transactions, while the older transactions do not see commits of new transactions. This transactional semantics avoids the difficult issues of: (*i*) rolling back an excessive amount of network state on aborts, and (*ii*) knowing when to commit a (large) transaction. As a result, our expectation is that network performance will be the same or marginally reduced. Note that our proposed semantics is different from nested transactions [14]. The contributions of this paper are as follows:

1. We identify a new type of bugs and their root cause—inconsistent packet processing. (§2)
2. We examine the possibility of using traditional database semantics for SDN packet processing. (§3)
3. We propose the multi-commit transactional model, a first step toward addressing the problem. (§4)

2. INCONSISTENT PACKET PROCESSING

In this section, we describe a new class of controller bugs we tentatively call "inconsistent packet processing" bugs. We first illustrate the problem on two examples.

Loop installation bug. Consider an OpenFlow controller that installs a lowest latency path between the source and destination on a switch-by-switch basis. The straightforward implementation of such a controller contains a potentially harmful race bug: if the topology changes in the middle of path installation, the controller will install the rest of the path according to the new routing. This can potentially lead to a forwarding loop.

Assume host $h1$ sends a packet to destination host $h2$ in the topology of Figure 1, with the $s1-s3$ link initially being down. Then, the following sequence of events will result in a forwarding loop in the network:

1. $h1$ sends a packet to switch $s1$, which sends it to the controller (because it has no rule for it);

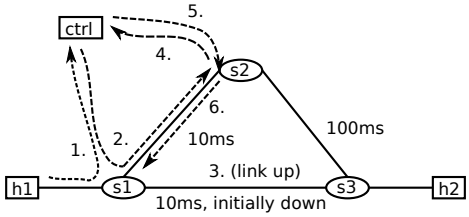

Figure 1: A loop formed by a buggy controller that installs paths on a switch-by-switch basis.

2. the controller installs rule $s1 \rightarrow s2$ (only way to reach $h2$) and sends the packet along the path;
3. meanwhile, the low-latency link $s1 - s3$ comes up;
4. $s2$ sends the packet to the controller (no rule);
5. the controller installs rule $s2 \rightarrow s1$ (the shortest path is now $s2 \rightarrow s1 \rightarrow s3$) and sends the packet;
6. rule $s2 \rightarrow s1$ is installed, thus creating a forwarding loop between $s1$ and $s2$;

Note that in a real deployment OpenFlow switches will refuse to forward packets back to their ingress port and simply drop them, thus creating a black hole instead of a loop. One may wonder if this simple check (either in the controller or in the switch) would prevent loops. Unfortunately, a slightly extended version of the example will create loops even if such checks are present as shown in the appendix.

"Rule amplification" bug. The second example involves `l2_multi` (a component of the popular POX platform), a multi-switch version of the learning switch that maintains a mapping $host \mapsto (switch, port)$ for every switch in the network. The controller code is involved in locating a destination host when a packet is directed to it. Specifically, when the host location h_{dst} is unknown, the controller instructs the first switch that encounters a packet for h_{dst} to flood the packet along a spanning tree. As flooded packets reach the adjacent switches, the controller instructs them to continue flooding the packet. Once host h_{dst} is reached and responds with a packet of its own, the controller sees the reply packet and learns h_{dst}'s location. However, the remains of the flood could still be traversing the network and when these duplicate packets reach OpenFlow switches, they are once again sent to the controller. Because the destination h_{dst} is now known, the controller routes these duplicate packets towards h_{dst}. Depending on the number of duplicates (which reflects network size) and topology, this may potentially generate a packet storm.[2] Moreover, the controller installs the (unnecessary) additional rules in the switches. Combined, these two effects ultimately hurt the performance of other flows in the network, as well as the overall network scalability for two reasons. First, additional rules occupy precious TCAM space. Second, the intra-switch bandwidth for rule installation can be surprisingly small [4]. This phenomenon manifests itself anytime the controller does not know the end-host's location, such as during controller startup, host mobility or periodic timeouts.

Generalizing the bugs Both of our examples contain a race condition. If we examine it more closely, the root

[2] Similarly to the previous example, the packet storm might be mitigated by OpenFlow switches. If the shortest path from flooded packet to h_{dst} points back to the link the packet came from, the OpenFlow switch will drop the packet unless `OFPP_IN_PORT` is used instead of a port number.

cause of the problem is what we term *inconsistent packet processing*: *i.e.*, the change of controller state while packets traverse the network. More precisely, both mentioned controllers were written with the assumption that the subsequent `PacketIn` handler executions for the same packet will be handled with a state that was left from the last time a packet or its duplicate was seen Unfortunately, this assumption does not hold. More importantly, designing controller programs which do not require this assumption is hard. In the lowest path delay example, after receiving a topology change event, the controller would need to invalidate not only the inaccessible paths, but also all partially-installed paths. Similarly, POX's `l2_multi` module would need to distinguish between copies of the flooded packet, and either drop or flood them.

These bugs can have severe consequences. Once created, the forwarding loop caused by the first bug stays in place until the rules expire. When the rule amplification problem manifests itself, on average 30% more rules get installed in our experiments (Section 6).

3. ACID TASTE OF CHALLENGES

The cause of inconsistent packet processing is the absence of *atomic* processing of events related to the same packet. In both examples, the controller logic assumed *isolation* from unrelated events while the chain of related `PacketIn`s was processed. These findings suggest examining the classic ACID transactional model.

Defining transactions and their semantic. Intuitively, a transaction is a set of related events which are to be processed in isolation from other events. However, unlike traditional databases, in SDN there is no explicit notion of which controller operations are related and should be processed as a transaction. Requiring the programmer to specify which events belong to which transaction would be hard because the events in the network are asynchronous in nature and can cause events to occur at the controller at different points in time. Rather, we strive for a solution in which the controller framework knows which events it should put together as a transaction on behalf of the programmer.

In the rest of the paper, we will concentrate only on `PacketIn` events.[3] We state that a set of `PacketIn` events forms a transaction (*i.e.*, the events are related) iff they are all events caused by a single packet and/or its duplicates. Work in that direction is however orthogonal to this paper.

Commit! But when? We do not require the programmer to explicitly specify the end of transactions. Instead, they end implicitly when the last packet associated with the transaction leaves the network. Thus, we now face the issue of identifying such events.

Rolling back transactions. Assuming we know when the transaction ended and should be committed, we need to take care of potential aborts, and rollback the modified state as if the transaction never happened. Note that state comprises not only the state of the controller, but also the state of the network. Unfortunately, rolling back the state of the network, especially the packets sent from the network edge, is infeasible.

The cost of enforcing ACID. The biggest problem in trying to provide the ACID semantics is the fact that the network cannot undo the packets that have been delivered to end-hosts. Thus, providing atomicity and isolation comes at a *high cost*: buffering every change to the controller and network state including the switch rules being installed, and the packets associated with the transaction. The location where this additional data would have to be stored is orthogonal to our discussion. A further problem is that buffering `PacketOut` commands (which instruct switches to emit packets) causes additional issues: if a packet was eventually destined to reach a switch without any matching rule, we will not receive the consequent `PacketIn` event, rather, we need to emulate it. When it comes to dealing with packets destined for end-hosts, there are two possible approaches for providing full ACID semantics: (*i*) have a shadow (emulated) copy of the network and controller for each transaction; when the transaction commits, copy the updated state back to the original controller (including packets sent over the edge links of the shadow network), (*ii*) use the real network, but have the ability to intercept packets at the edge so they can be dropped if a transaction aborts. The first approach is not only expensive but increases the "inconsistency surface" as the state of the network now needs to be synchronized with the state of the shadow network.[4] The second solution requires nontrivial capability at the edge links and introduces additional latency as packets would need to be delayed until the transaction commits.

4. MULTI-COMMIT TRANSACTIONS

To overcome the challenges of introducing the transactional model to SDN, we propose multi-commit transactions. The main idea is to relax the atomicity requirement. Instead of treating transactions as a single block of operations (which is either executed or aborted as a whole), we leverage the already present division of operations into different events. This split provides a sweet spot between proper atomicity and no atomicity at all. In essence, we treat every transaction as a set of subtransactions, with each subtransaction corresponding to one event. The commit/abort question is decided for each subtransaction separately and does not affect the previous decisions. If a subtransaction aborts, the state of the transaction rolls back to the state after the last successful (subtransaction) commit. Subsequent subtransactions will continue from this state and will be able to commit (unless another conflict occurs).

This model fits SDN because it solves both important challenges: (*i*) how to buffer the transaction data so that we can rollback easily, and (*ii*) deciding when to commit. We simply buffer the current event's actions until the end of the event handler and then commit. This is in contrast with a full ACID solution where we would need to buffer all actions and packets of all the event handlers called during the whole length of the transaction. Moreover, with multi-commit transactions there is no need to know when the transaction ends because we commit after each event. After an appropriate timeout, we may however garbage-collect the transaction metadata as there will be no subsequent subtransactions. In the rest of this section, we describe how we can provide good consistency properties and what the exact semantics of our model is.

Dependencies between transactions. The main reason

[3]The controller processing a packet for which a switch did not have a forwarding rule.

[4]For example, consider a rule expiring in the real network, while being still active in the shadow network.

for relaxing atomicity was to avoid expensive packet buffering. This leads us to a new problem—any packet leaving the network may potentially cause a response, *i.e.*, traffic that is causally related to that packet. A response packet will start a second transaction that should be processed in isolation. Isolation however breaks the dependency relationship (causality) and may result in conflicts between transactions.

We illustrate this *isolation problem* using the same `l2_multi` controller from section 2. This time, we assume it uses transactions, *i.e.*, the controller will process (flood) clones of the original packet in isolation from other packets. The first packet from h_1 to h_2 starts transaction T_1 which learns h_1 and floods the packet and its duplicates throughout the network. When host h_2 responds, the reply packet creates a new transaction T_2. Now consider that the original packet was still being flooded when the reply packet is observed, *i.e.*, T_1 did not finish. Assuming perfect isolation of controller state between T_1 and T_2, T_2 learns about h_2's location but does not know about h_1 and thus floods. Although flooding in both directions is harmless, at commit time we will need to decide the state of both transactions. Should we commit both of them even if they are in a read-write conflict $T_1 = [read(h_1), write(h_2)]$ versus $T_2 = [read(h_2), write(h_1)]$?

In this case, committing both transactions does not create an inconsistency. However, ignoring read-write conflicts or employing an eventual consistency model in general does not work for all controllers.[5]

To preserve dependencies, we need to take care of the state that other (possibly concurrent) transactions see after each subtransaction of T_i commits. Because a commit of subtransaction of transaction T_i might be accompanied by packets sent to the edge of the network, any transaction T_j starting after this commit must be able to see this commit, even if T_i and T_j continue concurrently.[6] On the other hand, old transactions should not be able to see modifications made by the newer transactions (*i.e.*, to ensure the isolation property). In summary, let T_1 and T_2 be transactions, with T_1 starting before T_2. Then, we would like the following behavior (x is a part of the controller state):

- T_1 and T_2 both read x: Ok.
- T_2 reads x after T_1 writes x and commits: T_2 will see the write (Dependency)
- T_1 reads x after T_2 writes x and commits: T_1 will not see the write (Isolation).

Guaranteeing consistency. To avoid inconsistencies in the controller, we need to provide serializability for the transactions.[7] In particular, this means that all commits (subtransactions) of a transaction must be ordered together. When coupled with the dependency criterion, this requirement forces the only possible serial order $T_{1,1}, T_{1,2}, \ldots, T_{1,n_1}, T_{2,1}, T_{2,2}, \ldots, T_{2,n_2}, T_{3,1}, \ldots$ where $T_{i,j}$ represents the $j-th$ subtransaction of transaction i and transactions are ordered by their starting time. This step also finalizes our "behavior table" by adding the following points:

- T_2 writes x after T_1 reads x and commits: T_2 should be able to commit.

[5] We include the example in the appendix.

[6] This violates the traditional isolation property but it is required in our weak atomicity model.

[7] We assume that a controller that is processing transactions serially is correct as it will avoid race conditions.

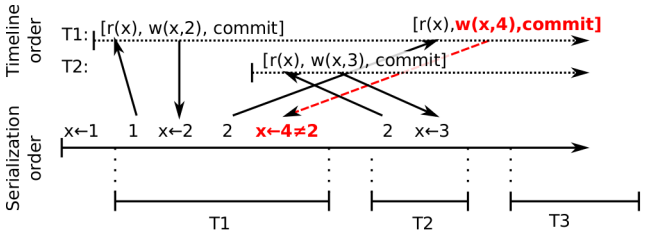

Figure 2: Illustration of the interaction between transactions. T_1's write conflicts with already committed T_2's read. T_1 must be aborted otherwise it could create an inconsistency.

- T_1 writes x after T_2 reads x and commits: This is a read-write conflict and one of the subtransactions needs to abort. As T_2 is already committed, T_1 needs to abort

Committing/aborting. All reads and writes in a transaction are logged. At commit time, the transaction T_i is able to commit either if (i) all pending operations are reads, or (ii) there is no read-write conflict in the controller. Although a write-write conflict of subtransactions does not necessarily need to be aborted, a write-write conflict spanning several subtransactions might cause inconsistencies. In our current model, we simply convert all write-write conflicts to read-write conflicts by requiring transactions to first read variables that are being written.

In the case a subtransaction aborts, we need to rollback its state. Fortunately, aborting a subtransaction is easy—because a subtransaction consists only of a single event handler, we can simply buffer all the controller state changes and network commands till the end of the handler. When the subtransaction commits, the buffers are flushed and in case of abort the buffers are cleared.

The possibility of some subtransactions aborting might lead to a concern that this model can cause additional programmer effort and/or that the transaction itself might be left in an inconsistent state. Here, we simply argue that aborting a `PacketIn` event is equivalent to not receiving that event in the first place. So, from the standpoint of the controller, an aborted subtransaction is equivalent to a lost packet—a situation that the controller should anyway be able to handle.

Moreover, subtransaction aborts are rarely necessary. Specifically, all read-only transactions are abort-free. Additionally, assuming a sequential execution of event handlers, all transactions that write only in the first subtransaction are also abort-free. We argue that most of the controllers are performing most of their writes in the first subtransaction and then they just read. Even if they continue writing state, it is usually just rewriting the state with the same values (*e.g.*, re-learning MAC mapping), which is efficiently equivalent to reads.

Finally, a careful reader might have noticed that in our case the aborted subtransaction always belongs to an older transaction. This might potentially lead to starvation. So far we do not have evidence of this behavior but we need to confirm this using experiments with more controllers.

Being optimistic. In addition to allowing our transactions to commit several times, we also require each subtransaction to be nonblocking. In particular, this requirement stems

from the fact that the transaction cannot block because of other transactions (*e.g.*, long-lived ones). Rather than rely on a locking mechanism, we advocate to use a form of optimistic concurrency control [11]. Optimistic concurrency control is especially useful in low-contention scenarios and we expect controllers to be such. Moreover, it ensures that read-only (sub)transactions will never fail.

5. PROTOTYPE

We have built an early prototype of OF.CPP for the POX controller platform written in Python. The goal here is to verify the feasibility of consistent packet processing. Though, the prototype is not optimized in any way. We leave the performance evaluation and any optimizations (especially in the transactional engine) for future work. Currently, the OF.CPP prototype does not buffer network operations, *i.e.*, it does not provide full event rollback functionality. This is mainly because the controllers we tested do not cause abort operations (the write happens only in the first event/step). Moreover, the prototype does not garbage collect transactions.

Usage. Using OF.CPP is simple, and we demonstrate this by providing the new transactional semantics in a drop-in replacement of the Python's dictionary. The user only needs to replace the controller's global state dictionaries with this new class. To relieve the programmer from specifying the transactions boundaries, we wrap the `PacketIn` handler with our own code.

Transactional dictionary. The main part of the prototype is `TransactionalDict`, the multi-commit transaction dictionary (key-value store). The dictionary API is consistent with the Python's API except for several new methods that are not visible to the programmer:

- `dict.newTransaction()`: creates a new transaction and returns its id.
- `dict.checkout(id)`: selects the currently active transaction. All read/write operations from now on will belong to this transaction.
- `dict.commit()`: tries to commit all pending changes of the current transaction. If there is a conflict, all pending changes are reverted and the transaction aborts (an exception is thrown).

The standard `dict[key]` read/write dictionary access operations are logged and appended to the pending list of the transaction.

Identifying transactions. To identify transactions, we store a "transaction tag" inside the packet header. Our current prototype uses the VLAN tag but it is easy to replace it with any other usable header field. Upon receiving an untagged packet, we create a new transaction and invoke the checkout operation. Upon receiving the tagged packet, we checkout the transaction with the id from the tag. Although we would need to intercept all commands to the switches that are related to the transaction, our prototype currently intercepts only the `PacketOut` commands.

6. EVALUATION

In this section, we examine: (*i*) the magnitude of the problem caused by the amplification bug in the POX `l2_multi` controller, and (*ii*) the applicability of our OF.CPP prototype to solve the problem. We set up an experiment in a Mininet [6] testbed with a Fat-Tree topology containing 20 switches and 16 end-hosts. Because the Fat-Tree topology contains cycles, we enable the POX spanning-tree module, which disables some links for flooding. This however affects only packets forwarded to the special "flood" (`OFPP_FLOOD`) port and leaves normal traffic unaffected. The experiment consists of a replay of a short trace with seven flows, emulating cold start of the controller. We modify the controller to issue non-expiring rules and record the total number of rules installed in the switches during the experiment.

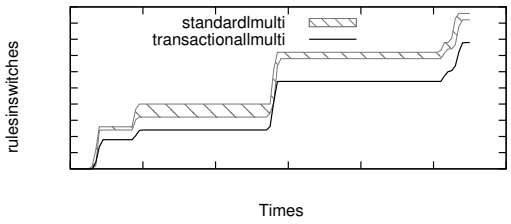

Figure 3: The number of rules installed by the standard and transactional versions of POX `l2_multi` in multiple trace replays in Mininet.

We compare this number against the one obtained by a transactional version of `l2_multi` running in our prototype, which avoids the rule amplification. The results in Figure 3 indicate that the rule amplification bug manifests itself across all experiments, albeit with varying intensity. On average, there are 30% more rules installed. Finally, the figure shows that OF.CPP behaves consistently. We manually confirmed that OF.CPP indeed installs only the expected rules.

7. DISCUSSION AND FUTURE WORK

We view this paper as a promising start to introducing a new transactional model into OpenFlow controllers. As the number of OpenFlow programmers increases in enterprise environments, we believe that OF.CPP will be effective in eradicating the difficult-to-diagnose kind of bugs we demonstrated in this paper. Next, we examine some potential questions:

How general is OF.CPP? To further demonstrate OF.CPP's generality, we plan to showcase its seamless integration into a few popular controllers. Doing so will require analyzing the data structures used by these controllers and developing their transactional counterparts.

What about other type of events? There are other events besides `PacketIn`s that might be important for the controller. As an example, consider link up/down events. Handling link up events in OF.CPP is easy as we can isolate the event from the old transactions. Link down events can be treated similarly if they do not conflict with concurrent transactions. However, if a link down event affects rules touched by a transaction, the transaction needs to abort. In this case, OF.CPP will need to inform the controller about the abort explicitly and the controller will then need to clean-up the state of previous sub-transactions similarly as in compensating transactions [10]. Although this scenario requires special controller recovery logic, we still believe that OF.CPP might have a value by helping to separate related events into transactions and identifying conflicts.

Is network performance going to suffer? We will

run detailed micro-benchmarks, especially to examine the effect of each planned optimization. We will also quantify the network-wide performance. That said, we anticipate marginal, if any, performance impact. For example, although the events will effectively be serialized, the "database" is having only one client, namely the controller. Multithreaded controllers will start to get deployed, and we will address the important issue of incorporating multithreading in our future work.

8. RELATED WORK

The most related work to ours is consistent updates [13]. The important difference is that while consistent updates are solving the network-consistency problem, OF.CPP solves consistency issues *inside* the controller. NICE [3] exercises different network event orderings, while NDB [7] collects packet traces. Although both tools can uncover consistency problems (NICE in a model, NDB in production), they can only report the existence of a bug and do not fix it. Header Space Analysis [8] and VeriFlow [9] check the data-plane. These tools can verify the current network configuration and thus can detect the controller inconsistency only through the data plane. Frenetic [5] programs are still written on a switch-by-switch basis. Pyretic [12] maps high-level policies to low-level rules. While this may protect high-level programmers from consistency issues, developers of the runtime system would need to deal with these issues. Our idea of multi-commit transactions is based on ideas from optimistic concurrency control [11] and nested transactions [14]. Unlike nested transactions, multi-commit transactions do not preserve atomicity and cannot be fully rolled back. Instead, by relaxing atomicity, multi-commit transactions can span long periods of time without blocking. To ensure consistency, we employ ideas from causal consistency [1] and serializability [2].

9. CONCLUSION

In this paper, we have described a new class of problems affecting SDN that stem from the lack of isolated packet processing in OpenFlow controllers. We have detailed two examples of this problem, that decrease network reliability and performance, respectively. After discussing the inability of directly applying the classic database techniques to mitigate this problem, we have proposed a new semantics that we refer to as multi-commit transactions. Finally, we have demonstrated the ease of incorporating this new model into POX, a popular OpenFlow controller.

10. REFERENCES

[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming, 1994.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman, 1987.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *NSDI*, 2012.

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[5] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *ICFP*, 2011.

[6] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *CoNEXT*, 2012.

[7] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *HotSDN*, 2012.

[8] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[9] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[10] H. F. Korth, E. levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *16th International Conference on Very Large Databases*, 1990.

[11] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[12] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *NSDI*, 2013.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[14] R. F. Resende and A. El Abbadi. On the serializability theorem for nested transactions. *Inf. Process. Lett.*, 50(4):177–183, May 1994.

APPENDIX

A. EVENTUAL CONSISTENCY IS NOT ENOUGH - NAT EXAMPLE

Here we include a simple load-balancer/Network Address Translator controller which proves that forms of eventual consistency are not enough. In particular, we show that cyclical read-write conflicts between two objects are dangerous.

In this example, we assume a single-switch OpenFlow controller emulating a NAT device which preserves address of remote clients (useful for servers which need to respond differently to different IP addresses, *e.g.*IP white/blacklisting). The controller stores the NAT mapping in the format `map[remapped_header] = original_header`.

This mapping is useful for installing the *reverse* flow when/if it comes. Note that the opposite mapping `map[original] = remapped` is not saved as it is unnecessary. The controller behavior is described in Fig. 4.

Normal controller operation For the sake of the example, let R denote IP address of the remote host, with the IP address of the server behind NAT being S and the external-facing (client-reachable) IP address of NAT as C. Consider the controller receiving a connection request (SYN packet) with header `R:1111->C:80` on the external link. The controller recognizes that the destination port 80 belongs to S (or simply picks S based on the load). The controller will then install a rule rewriting `R:1111->C:80` to `R:1111->S:80`. To install the rule in

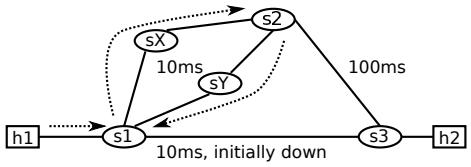

Figure 5: Extended loop example - there are two disjoint shortest paths between s_1 and s_2 and controller uses different one for each direction

```
map = {}

def opposite(Header h):
  # Swap source and destination
  return new Header(src=h.dst, dst=h.src)


def packetIn(Packet pkt):
  if map[pkt.header] is not None:
    # Opposite direction of already installed flow
    install_rule(rewrite(pkt.header, map[pkt.header]))
  else:
    # Rewrite address
    new_header = NAT(pkt.header)
    # store reverse mapping so we can identify
    # the opposite direction of the flow
    map[opposite(new_header)] = opposite(pkt.header)
    install_rule(rewrite(pkt.header, new_header))
```

Figure 4: Example NAT controller

the opposite direction (if the server responds), the controller stores `map[S:80->R:1111 ] = C:80->R:1111`.

When the web server replies with packet `S:80->R:1111`, the controller uses the mapping to install the rewrite rule.

Conflict Now, consider that the controller uses transactions to isolate unrelated events. If the original transaction was not committed when the response from web server arrived, the controller would see flow `S:80->R:1111` without any entry in the mapping `map`. Because there is no mapping, the controller considers this request to be a connection initiated by the web server to a remote endhost. As such, the controller will perform another address translation – it assigns an external-facing port and rewrites the header. Let as assume there will not be a conflict in port assignment (either because the assignment is random or based on the hash of the connection), for example let it use port `C:2222`. The controller installs rewrite rule *Rewrite*(`S:80->R:1111` => `C:2222->R:1111`) and sets `map[R:1111->C:2222 ] = R:1111->S:80` in order to be able to install the opposite rule later when the distant endhost replies. Now, if we commit both of these transactions, we have:

1. dictionary with two mappings, these mappings are mutually inconsistent but are not in the write-write conflict (transactions write to different dictionary keys)
2. flow from `R:1111->C:80` is rewritten (in the opposite direction) to `C:2222->R:1111` which efficiently breaks the connection.

B. LOOP EXAMPLE (EXTENDED)

Here we show that default OpenFlow protection against loops, *i.e.*OpenFlow switch dropping packet destined to the input port unless the port is explicitly `OFPP_IN_PORT`, is not enough to prevent loops. In particular, assume the same scenario as in Fig 1 but now with two more switches sX and sY. These switches provide two disjoint paths between $s1$ and $s2$ with equal latency as shown in Fig. 5. When the packet from $h1$ arrives, the controller installs rules $s1 \rightarrow sX$ and $sX \rightarrow s2$. After the controller received notification of link $s1 - s2$ being up, it decides to continue with path $s2 \rightarrow sY$, $sY \rightarrow s1$, $s1 \rightarrow s3$ creating a loop after first two rules.